Parallel I/O on Compressed Data Files: Semantics, Algorithms, and Performance Evaluation

Siddhesh Pratap Singh and Edgar Gabriel

Parallel Software Technologies Laboratory, Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: {spsingh, egabriel}@uh.edu

***Abstract*—Many scientific applications operate on data sets that span hundreds of Gigabytes or even Terabytes in size. Large data sets often use compression to reduce the size of the files. Yet as of today, parallel I/O libraries do not support reading and writing compressed files, necessitating either expensive sequential compression/decompression operations before/after the simulation, or omitting advanced features of parallel I/O libraries, such as collective I/O operations. This paper introduces parallel I/O on compressed data files, discusses the key challenges, requirements, and solutions for supporting compressed data files in MPI I/O, as well as limitations on some MPI I/O operations when using compressed data files. The paper details handling of individual read and write operations of compressed data files, and presents an extension to the two-phase collective I/O algorithm to support data compression. The paper further presents and evaluates an implementation based on the Snappy compression library and the OMPIO parallel I/O framework. The performance evaluation using multiple data sets demonstrate significant performance benefits when using data compression on a parallel BeeGFS file system.**

I. Introduction

Many scientific applications operate on data sets that span hundreds of Gigabytes or even Terabytes in size. These applications often spend a significant amount of time reading and writing input and output files. To cope with the challenges posed by file I/O, High Performance Computing (HPC) applications have long relied on parallel I/O APIs and libraries such as MPI I/O [27], HDF5 [19], NetCDF [9], or ADIOS [25]. These libraries offer a number of unique features that have contributed to their success, such as *collective I/O operations*, i.e. functions that allow an application to express file access of an entire group of processes semantically as a single, logical operation [11, 14].

A typical example for a data file used in an HPC application contains a multi-dimensional matrix, stored in binary format using e.g. row-major ordering. The dimensions of the matrix can easily be determined, and every row/column as well as every element have the same extent and size. If – as part of the data decomposition in the parallel application – a process is required to read/write a certain number of rows of the matrix, it can easily calculate the precise offsets into the file to access those elements. The ability of each process to calculate the offsets for the elements of interest is an essential requirement for parallel I/O: all processes can simultaneously start to read/write different parts of the file, without interfering (logically) with each other.

Large data sets are often compressed to reduce the size of a file, save space on a storage system, and to minimize transfer times when downloading a large file from a remote resource. Yet, as of today, parallel I/O libraries are not able to operate on compressed data files. Applications using compressed data files will have to decompress the file before the start of the simulation, and compress resulting output files after the simulation has finished. This is however highly inefficient, since sequential compression of large data files is very expensive. To give an example, compressing one of the data files also used later in the evaluation section of this paper – with 16 GB arguably a smaller data set using today's standards – with the popular gzip [17] tool took more than 28 minutes on an Intel Xeon E5-2680 processor.

A more performant solution would be for the parallel I/O library to be able read and write compressed data files, making sequential compression and decompression unnecessary, and allowing to take advantage of other features that parallel I/O libraries have to offer. This is however for multiple reasons highly challenging. The main difficulty stems from the fact that there is no (easy) calculation that can be performed to map the logical, uncompressed offset of an element in a file – i.e. what an MPI application would operate on based on the current MPI I/O interfaces and syntax – and the actual position of the element in the compressed data file. Furthermore, accessing a particular element might require reading/writing more than just the requested item.

This paper introduces parallel I/O on compressed data files, discusses the key challenges, requirements, and solutions for supporting compressed data files in MPI I/O. The key contributions of the paper are i) discussion of the impact on MPI I/O operations when using compressed data files, ii) detail handling of individual read and write operations of compressed data files; iii) present an extension to the two-phase collective I/O algorithm to support data compression; and iv) evaluate an implementation supporting parallel I/O on compressed data files based on the Snappy [8] compression library and the OMPIO [12] parallel I/O framework.

The remainder of the paper is organized as follows. Section II describes the requirements, limitations, and concepts required to use compressed data files for parallel I/O operations. Section III introduces the algorithms and techniques developed to support parallel I/O on compressed data files, and provides the most relevant implementation details. Section IV evaluates the performance of our solution on multiple data sets. Section V discusses the most relevant related work in the area. Finally, section VI summarizes the contributions and findings of the paper, and lays out the future work in this domain.

II. Goals, Requirements and Limitations

The main goal of this research is to support compressed data files in MPI I/O. MPI introduced the notion of parallel I/O in version two of the specification. In combination with parallel file systems, MPI I/O has been shown to significantly improve the performance of I/O operations compared to sequential I/O libraries. The most distinct features of MPI I/O include:

1) Collective I/O: the ability to abstract the file I/O operations of a group of processes as a single, logical operation;
2) File view: registering an access plan to the file in advance;
3) Relaxed consistency semantics: updates to a file might initially only be visible to the process performing the action;
4) Shared File pointers: manage a single file pointer for a group of processes.

To maximize the usability and impact of our work, we aim to further achieve this goal by taking the following requirements into account.

- **Minimize changes** required to an application for reading/writing compressed data files. For example, an application should be able to use the same code to specify a file view and the same syntax for `MPI_File_read` or `MPI_File_write` operations for a compressed file as it does for uncompressed files.
- **Maximize performance** of parallel I/O operations on compressed data files. For this, it is essential that the compressed output file is created in a single pass, and avoids a two step process, i.e. writing first an uncompressed file, and compress it in a secondary step. Similar requirements apply for reading a compressed data file.
- **Interoperability** with other tools/libraries supporting the same compression algorithm: a compressed output file created by the parallel I/O library has to be readable by another tool using the same compression algorithm or library. Similarly, a data file created and compressed by an external application/tool has to be a valid input file for the parallel I/O library. This requirement equates to the parallel I/O library being fully compliant with the specification of the compression algorithm.

As discussed earlier, parallel I/O operations rely on the ability of each process to i) calculate offsets into the file, and ii) execute file operations independently of each other. The first item also poses the main challenge in this work, since it is not easily possible to map a logical, uncompressed file offset – i.e. what e.g. an MPI application would operate on based on the current interfaces and syntax – into an actual offset in the compressed file. Consider for example, process 0 trying to write 64KB of data at the beginning of the file (offset = 0), and process 1 trying to write 128KB of data right after the data of process 0, namely at (logical, uncompressed) offset 64KB. If no compression is applied, the second process can seek its file pointer to the correct position in the output file, and both processes can write their data simultaneously. This is however not possible with compression. Process 1 does not know the size of the compressed buffer of process 0. If the data of process 1 is expected to follow in the file the data of process 0 without leaving a gap, process 1 can only perform the write operation after process 0 has finished compressing its data, and either finished writing its data to the file, or communicates the size of its compressed buffer to process 1.

In order to support parallel processing on a compressed data file, the compression algorithm has to support concurrent compression/decompression of different portions of the file, typically by operating on a chunk-by-chunk basis. Compression libraries such as Snappy [8] or bzip [18] fulfill this criteria, while some other libraries such as 7z [35] do not. Most compression formats do not tolerate gaps in the compressed file. This restricts on how data can be written and what is considered a valid access pattern when writing a file. (Note, that if one would lift the interoperability requirement, one could allow for gaps in the compressed file. This could be explored at later stages of our work e.g. by developing a new *internal data representation* for the MPI I/O library).

This limitation (no gaps in the resulting file) and performance considerations introduce two restrictions on the parallel I/O operations invoked by an application operating on compressed data files.

- **Monotonically increasing offsets for write operation**: The offsets used into the file have to be monotonically increasing per write call. It is thus not allowed to rewrite parts of an existing file. Since the compression ratio depends on the actual data to be compressed, the 'new' part of the file will typically have a different compressed length than the original one. Since no gaps are allowed in the resulting output file, the entire remainder of the file would have to be relocated/rewritten, which is not an option for large files. For very similar reasons, it is also not allowed to leave a gap in the file with a write operation, and fill it with a subsequent write operation.
- **Individual write operations are only permitted from a single process** : When using individual I/O operations, each process performs its own write operations without coordinating with other processes. Since the length of a compressed block is not known except for the process performing the compression, having multiple processes performing individual write operations would inadvertently lead to gaps in the file, or processes overwriting each others data. Note that there are alternative interface that allow multiple processes to write data simultaneously, namely collective I/O operations and shared file pointer operations.

There are no restrictions on read operations. Collective write operations are conceptually supported as long as they fulfill the first requirement (monotonically increasing offsets). While some of these limitations seem restrictive compared to the flexibility offered by the MPI I/O specification for non-compressed data files, it should be noted that the Hadoop Filesystem [26] (HDFS) has very similar restrictions regarding re-writing files and the offset requirements (one can only append to an existing file in HDFS), and it did not seem to hinder most applications. Furthermore, the fact that MPI I/O specification already included a mode where only one process is allowed to access the file (`MPI_MODE_SEQUENTIAL`) indicates that these type of restrictions have been anticipated for certain environments and have been considered acceptable. The user still has the ability to fully exploit multi-process parallel I/O for this scenario by using collective I/O operations or shared file pointer operations.

III. Concept and Implementation Details

In the following we present the conceptual and algorithmic solutions for supporting compressed data files as well as the most relevant implementation details.

A. Individual I/O

Individual file I/O operations are defined in MPI I/O to be executed by each process separately and without any coordination across processes. In contrary to the collective I/O operations discussed in the next subsection, a process executing an individual read or write operation does not have the ability to determine whether another process is also executing a file I/O operation at the same time. Conceptually, supporting individual write operations for compressed data files requires i) allocating a temporary buffer to compress a chunk of the user provided input buffer; ii) compressing the data; iii) writing the compressed chunk to file. An I/O library might be able to overlap compressing one chunk with the file write operation of the previous chunk by using multiple threads.

Individual read operations will require a similar approach: the parallel I/O library will have to allocate enough temporary buffer to read in an entire chunk of the compressed file, read the chunk(s) from the compressed data file that contains the requested data items, decompress a chunk, and copy the requested data from the temporary buffer into the buffer provided by the user code for this operation. A crucial aspect to increase the performance of read operations will rely on the ability to quickly identify the part of the file that contains the data items requested by the user. Recall that the user codes provides the (`offset, length`) information for the read operation in terms of uncompressed data (since this is the semantics used e.g. in MPI I/O). The parallel I/O library will have to locate the location of that starting byte in the compressed data files given the uncompressed offset. To accelerate this operation, we introduce an *annotation file*, which maintains a list of (`uncompressed offset, compressed offset`) tuples for each chunk (or every n th chunks) in the data file. This allows to restrict the search for a particular offset to i) scanning of the annotation file (which is orders of magnitudes smaller than the data file itself) and ii) reading a subset of the compressed data file. The annotation file will be automatically generated by the parallel I/O library when writing a file. If the user provides however a data file that was created and compressed by another library (e.g. a Spark application), a stand-alone tool can be used to generate the annotation file.

B. Collective I/O

Collective I/O operations represent higher level APIs which allow to reorder data across processes to match the layout of the data on the file system level. Consider as an example a scenario, where four processes operate jointly on a matrix stored in a file. The overall matrix size is 128×128 elements, and each process holds a subset of the matrix of 64×64 elements. If the 2-D matrix is stored using a row-major ordering, the data stored in the file does not match logically the data distribution across the process, since the 128 elements of each row are distributed across two processes. Collective I/O operations allow to re-arrange the data among the processes for read and write operations to match the layout of the data on the file system. Thus, it allows to decouple the data decomposition used on the process level from the data layout on the file system level.

The most widely used algorithm for collective I/O is the two-phase I/O [34] algorithm, which – as the name implies – consists of two steps. For a write operation, the first phase redistributes data among the processes to match the layout of the data on the file (also referred to as the shuffle step), while the second phase executes the actual write operation. The redistribution is necessary, since most file systems impose a significant performance penalty if data is not written in the same order as it is stored in the file.

The two-phase I/O algorithm introduces two further optimizations. First, only a subset of the MPI application processes actually touch the file, i.e., perform read or write operations. The processes executing file I/O operations are also referred to as the *aggregators*. Second, for very large collective read and write operations, the two-phase I/O algorithm splits the data and performs the operation internally in multiple cycles. This keeps additional memory requirements on the aggregator processes within reasonable limits, and allows for potential overlap of the shuffle step and the write operation of subsequent cycles.

For collective I/O writing compressed data files, the implementation is still based on the standard two-phase I/O algorithm, with a number of additional challenges. In the following, we will only highlight one aspect, namely whether the compression of the user provided data shall be done on each individual process before communicating with aggregator processes, or whether to perform that operation on the aggregator processes. The second option seems initially more intuitive, since data from multiple processes is assembled based on their offset in the file by the aggregator. This approach has however one major downside. Recall that in the two-phase I/O algorithm, each aggregator is responsible for writing $1/n_a$ th of the overall data to be written. For example, if the total amount to be written in a single `MPI_File_write_all` function call is 12GB, and four aggregators are being used, the first aggregator will write bytes $0-(3GB-1)$, the second aggregators will write bytes $3GB-(6GB-1)$, etc.. Since one of our previously discussed restrictions is that the output file cannot contain any gaps, the second aggregator will have to wait on the first aggregator to report the overall size of its compressed buffers in order to determine its starting offset into the file. This would lead to serialization of the aggregator operations **or** require aggregators to hold a copy of the entire compressed data that they have to write in memory before starting the actual write operations, which is not feasibly in most cases.

When using the first option, each process compresses their own data locally before the shuffle step performed as part of the two-phase I/O algorithm. While this approach will also increase the memory footprint of every process since they have to hold the entire compressed buffer in memory before starting the data transfer to the aggregators, the additional memory requirements are still more manageable than in the second scenario. Furthermore, it increases the level of parallelism for the data compression. The downside of this approach however is that it can lead to a larger number of smaller 'compressed chunks' compared to the second option.

Algorithm 1 provides the logical operations occurring in the two-phase collective write operation using data compression. Note, that elements enlisted in curly brackets { } represent arrays in this annotation. The input to this algorithms is a buffer in the main memory of each processes, a derived MPI datatype that describes the layout of the data in the memory of a process, and an MPI file handle. The MPI file handle will have a file view attached to it, which provides a description of which parts of the file a process will write to.

At the beginning of the algorithm, each process flattens the derived datatype describing its data layout in memory into a vector of memory addresses and length, and matches it with the file offsets obtained by flattening the derived datatype stored as the file view, providing a description of where each chunk of data will have to be written (line 1). Each processes determines the total number of bytes written as part of this collective I/O operation (line 2), and the overall amount of data written across all processes (line 3). The total number of bytes written and the product of the number of bytes written per cycle and the number of aggregator processes is being used to determine the number of internal cycles it will take to execute this collective write operation (line 4). The smallest and largest file offsets across all processes are being used to identify the part of the file that is being modified during this file_write_all operation, and each aggregator has a subset of this file domain assigned to it (line 5). Using multiple All-gather(v) operations, a global sequence of file offsets and length are being constructed, allowing each process to calculate how many bytes of data they have to contribute in each cycle to each aggregator (lines 6-8).

Algorithm 1 Two-Phase Collective Write Operation with Data Compression

```
Require: memory buffer, count, datatype, file handle, file view, NumberOfAggregators, BytesPerCycle
1: Convert file view, memory buffer, count, datatype into a vector of $\{MemoryAddress, FileOffset, Length\}_k$ on every process $k$
2: Calculate number of bytes $DataOnProc_k$ to be written by process $k$
3: $TotalData \leftarrow \sum(DataOnProc_k)$ (Allreduce)
4: $NumberOfCycles \leftarrow \lceil TotalData/(BytesPerCycle \times NumberOfAggregators)\rceil$
5: Calculate file domain for each aggregator $j$.
6: Construct global vector of $\{FileOffset_g$, $Length_g$, $PocessId_g\}$ as a union of $\{FileOffset, Length\}_k$ of every process $k$ (Allgatherv)
7: Sort global vector $\{FileOffset_g, Length_g, ProcessId_g\}$ based on the $FileOffsets$.
8: Determine on every process $k$ the vector of $\{FileOffset', Length'\}_i^j$ for the data process $k$ needs to send to aggregator $j$ in cycle $i$.
9: Compress every buffer belonging to each tuple $\{FileOffset', Length'\}_i^j$ separately.
10: Send $\{CompressedLength\}_k^j$ from process $k$ to aggregator $j$
11: if $ProcessId\ is\ aggregator$ then
12:   for k=0 to NumberOfProcesses do
13:     Recv $\{CompressedLength\}_k$
14:   end for
15:   Use $\{CompressedLength\}$ to recalculate $\{FileOffsets_g\}$ using the same order of elements as determined in line 8 and replacing every corresponding entry of $Length_g$ with the new value $CompressedLength$
16:   Calculate compressed starting offset for each aggregator domain (Exscan)
17: end if
18: for i=0 to NumberOfCycles do
19:   for j=0 to NumberOfAggregators do
20:     Send $Data_i^j$ to aggregator $j$
21:     if $ProcessId\ is\ aggregator$ then
22:       for k=0 to NumberOfProcesses do
23:         Recv data from process $k$
24:       end for
25:       Write Data to disk
26:     end if
27:   end for
28: end for
```

The key additions to the standard algorithm are given in lines 10 – 18. After determining which parts of its buffer have to be sent to each aggregator in a given cycle (line 8), each process compresses its data following this partitioning (line 9), and communicates the length of the compressed buffers to the aggregators (line 10). An aggregator consequently has to update its own calculations, replacing the original, uncompressed length with the compressed length for the same data item(s), and recalculate the offset where the data will be written using the compressed length (line 15). The order of the data items in the compressed file will still be based on the order obtained by sorting the uncompressed offsets in line 7. Once every aggregators has updated its list of data items that it has to write with the compressed lengths, it can use the largest offset value plus the compressed length of the last item to calculate the starting offset for the next aggregators. This last step can be implemented efficiently in MPI using the `MPI_Exscan` function. Lines 17-24 perform than the data aggregation step in each cycle, followed by the system level write operation of the aggregators (line 25).

Overall, the extended two-phase collective write algorithm requires some additional communication of meta-data such as offsets and length (lines 10, 13, and 16). However, the communication of actual data buffers (lines 20 and 23) occur on the compressed data buffers, which will in most circumstances be smaller than the uncompressed buffers.

C. Shared file pointers

In contrast to individual file pointer operations discussed so far, where the position of a file pointer is only dependent on sequence of file I/O operations of a particular process, the position of the shared file pointer is based on the sequence of file I/O operations of the entire group of processes. Common use-cases for shared-file pointers include dynamic workload distribution (i.e. each process will fetch the next data item from a shared input file), or writing a parallel log-file [22, 24].

Shared file pointers require an entity to manage the current position of the shared file pointer. On file systems that support file locking, this can be implemented by storing the position of shared file pointer in a separate file, and use the file systems locking mechanism to restrict access to this file to only one process at a time. Thus, access to the shared file pointer file is serialized, while access to the actual data file might still occur in parallel. Supporting compressed data files for shared file pointer operations require an extension to the format of the shared file pointer file: instead of storing the current position of the shared file pointer in the file, the parallel I/O library will have to store the logical (uncompressed) offset into the file as well as the actual offset into the compressed file.

D. Implementation details

To evaluate the practicality and the performance of the solutions described in the previous subsections, we have developed an implementation supporting compressed data files in MPI I/O. The implementation is based on the OMPIO [12] parallel I/O components of Open MPI and the Snappy [8] compression library. The Open MPI project [16] is an open source implementation of the MPI specification that is jointly developed and maintained by a consortium of academic, research, and industrial partners. The library is built around the Modular Component Architecture (MCA), which allows a compile time or runtime selection of the components used by Open MPI. Each major functional area in Open MPI has a corresponding back-end component framework, which manages its own modules. This concept is used for example to seamlessly support multiple network interconnects, or different implementations of collective communication operations.

The same concept is exploited by OMPIO [12] to provide a highly flexible parallel I/O infrastructure. OMPIO is implemented as a module of the `io` framework of Open MPI. Upon opening a file, the OMPIO component triggers the MCA selection logic for multiple sub-framework, namely:

- The *file system framework (fs)*: abstracts general file manipulation operations, such as opening, closing, and deleting a file.
- The *file byte-transfer layer framework (fbtl)*: provides the abstraction for individual read and write operations.
- The *file collective I/O framework (fcoll)* : provides the abstraction for collective file I/O operations.
- The *shared file pointer framework (sharedfp)* : abstracts functions for managing the shared file pointer.

Each module within the component framework is an independent unit which can configure, build and install itself. More than one module may exist within each framework and multiple modules from a framework may be used within the runtime of an Open MPI program.

Snappy [8] is a compression/decompression library developed by Google and widely utilized and deployed in the Data Science community e.g. within Hadoop and Spark [36] applications. Snappy aims to maximize compression and decompression speeds while maintaining reasonable compression ratios. The Snappy file format fulfills the requirements described above for parallel I/O: by default, Snappy operates on 64KB chunk sizes [6]. Furthermore, the Snappy file format defines that a concatenation of multiple Snappy compressed files represent a valid Snappy file [6].

In order to support reading and writing a Snappy compressed file using MPI I/O functionality, we developed multiple components supporting the required operations, namely:

- *composix*: an fbtl component supporting individual read and write operations of Snappy compressed files. The component is dervied from the standard *posix* fbtl component used by OMPIO on most parallel file systems.
- *vulcomp*: an fcoll component supporting collective read and write operations of Snappy compressed files. The component is an extension of the *vulcan* collective file I/O component, which is used by OMPIO on all file systems except for Lustre.
- *clockedfile*: a sharedfp component supporting reading and writing Snappy compressed files using a shared file pointer. The component is based on the *lockedfile* sharedfp component, which requires file system support for file locking.

These components are used in the subsequent performance evaluations. In order to read or write a Snappy compressed file, a user could use the standard Open MPI mechanism to force the selection of these components, e.g. using additional arguments to the mpirun command, or environment variables. While this works, this approach has the downside that **all** files used by that application will be forced to use these particular components. To support selectively using compression on some files but not on others, the user can provide a new info object to the `MPI_File_open()` function requesting data compression. The info object will force the framework selection logic to select the new fbtl, fcoll, and sharedfp components supporting Snappy compression.

It has been verified that an output file created by these components can be correctly decompressed using the snzip tool [5], a sequential compression/decompression tool based on the Snappy algorithm, and vice versa, a file compressed using the snzip tool could be read by the new components.

IV. Performance Evaluation

In the following, the performance of parallel I/O operations for compressed data files is being evaluated for various data sets. The focus is on collective write operations, since it represents i) the biggest difference and highest benefits compared to sequential compression tools, and ii) since write operations are significantly more complex than read operations.

The tests have been executed on the *crill* cluster at the University of Houston. The cluster consists of 26 nodes with approx. $1,000$ cores, and a QDR InfiniBand network interconnect. The cluster has an NFS as well as a parallel BeeGFS v7.0 file system [7] distributed over 16 I/O servers, the latter being the used for the subsequent tests. The BeeGFS file system uses a stripe-size of 1 MB. The parallel I/O library used is based on the master repository of Open MPI, using gcc version 4.8.5 and the Snappy compression library version 1.1.7. All test have been executed at least three times, and the minimum values across the measurement series is being presented in the subsequent subsection, unless explicitly denoted differently. Since the cluster was used in a dedicated mode for these measurements, the variance between the individual runs was less than 1% in most cases, and is being omitted for the sake of clarity. The serial tests are run on a single node using the snzip [5] compressor on the exact same dataset as the collective tests.

A. Matrix tests

The first set of tests are based on a 2-D matrix of double precision floating point values. The matrix values are chosen semi-randomly from a bucket containing a fixed number of floating point values, increasing the chances of a value to be picked if it has not been used for a certain number of requests. The elements in the bucket are generated using a random number generator at the beginning of the test. The size of the bucket enables us to control the compression ratio of the matrices, and is used later in this section to evaluate the impact of this parameter on the performance of the file I/O operations.

The matrix size is determined by a user provided value and the number of processes. For a user provided value of n and p processes, each process holds a matrix of size $n \times n$ of 8-byte floating point values. The overall matrix size is thus $n \times n \times p$ elements.

Two different data decomposition are explored as part of the analysis, namely a one dimensional and a two dimensional decomposition. For the 1-D case, each process holds a contiguous number of rows of the matrix, while with 2-D decomposition each process holds a square block of the overall matrix. In contrary to the 1-D distribution, every row of the global matrix is distributed over multiple processes. The data distributions are implemented in the test code using MPI derived datatypes and file views. For the sake of uniformity across these tests, all process counts used subsequently are square number, since this simplifies the 2-D data decomposition significantly.

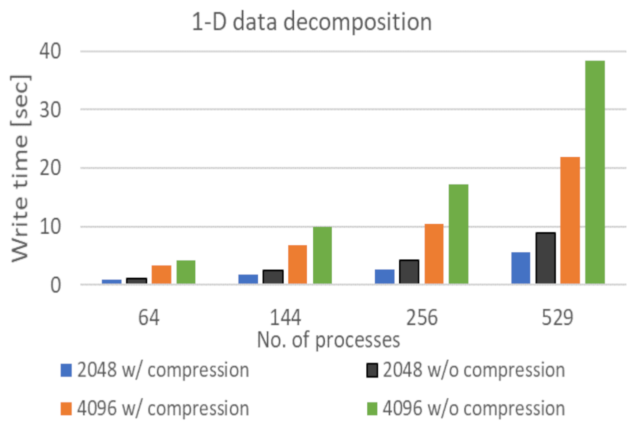

Fig. 1. Performance of a collective write operation for various process counts and matrix sizes with and without data compression for a 1-D data decomposition.

Tests have been executed for 64, 144, 256 and 529 processes distributed equally on 16 compute nodes. The benchmarks measure the time required to write the matrices to the BeeGFS file system using `MPI_File_write_all`, with and without compression. Data shown are for two local matrix sizes, namely each process holding 2048×2048 elements of the matrix, and 4096×4096 elements. Since the overall matrix size scales with the number of processes used, the overall file sizes used in this set of tests varies between 2GB for the smallest test case presented here and 66GB for the largest one. For this set of tests, the size of the bucket used to populate the matrix has been set to $1,000$ elements, resulting in a compression ratio (i.e. the ratio of the uncompressed vs. compressed file size) between 1.55 and 1.8 and a size reduction of around 38 - 44% due to compression.

Fig. 1 shows the results obtained for the 1-D data distribution. The results indicate significant performance improvements when writing a compressed output file for all test cases, performance benefits were more pronounced for larger file sizes and process counts. The performance improvement of writing a Snappy compressed output file vs. an uncompressed one ranges from 17% for the smallest test case, up to 43% for the largest test case.

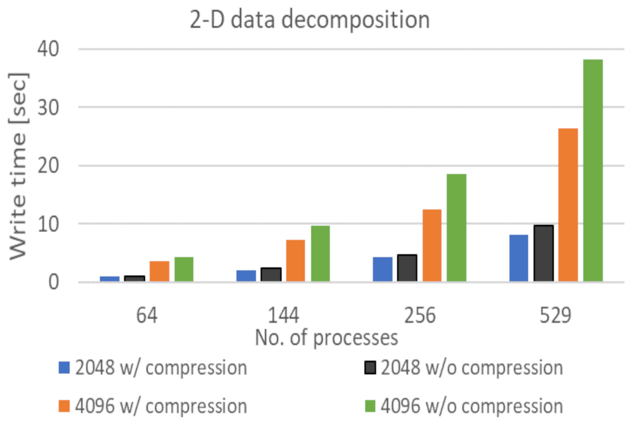

Fig. 2. Performance of a collective write operation for various process counts and matrix sizes with and without data compression for a 2-D data decomposition.

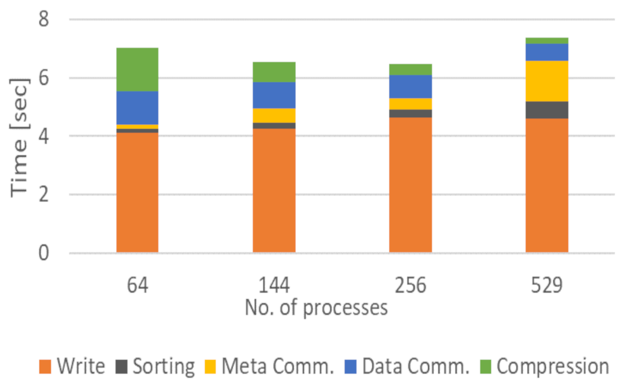

Fig. 3. Breakdown of the individual components of a collective write operation for a fixed problem size and different process counts.

The results obtained for the 2-D data decomposition are shown in fig. 2. While the overall trend is similar to the results obtained with the 1-D test cases, there are some notable differences, which can be attributed to the fact that the compression ratio obtained for the 2-D test cases are lower than the 1-D test cases. Recall that the key difference between the 1-D and the 2-D data decomposition is that a single row of the global matrix is stored on a single process in the first case, but not in the second one. As a consequence, all data held be a single process in the 1-D test case is contiguous in the file, and can thus be compressed into equal chunks using the maximum chunk size supported by the Snappy library, i.e. 64KB. That is not the case with the 2-D decomposition. Assuming for example 64 processes and each process holding 4096×4096 elements of a matrix, a single row of the global matrix will consist of $4096 \times \sqrt{64}$ elements. Since data is compressed by each process individually (before aggregation), and is restricted by the number of elements contiguous in the file, the 2-D decomposition leads inadvertently to more but smaller chunks than the 1-D decomposition. This leads ultimately to the lower compression ratio and thus to somewhat lower performance benefits compared to the 1-D case. Nevertheless, our tests show even for the 2-D test cases performance improvements up to 32% when using data compression.

1) Timing Breakdown: To gain a better understanding of the internal operations of a collective I/O operation, we instrumented our code and measured the time spent in individual sections. This set of tests are using the 2-D data decomposition, since they are representative of a large number of parallel applications. Matrix sizes have been chosen for the various process counts to generate a (nearly) constant size output file, by scaling the dimensions of the matrices to match the 529 process test case with a local matrix on each process of 2048×2048 elements.

The individual times and code sections monitored are:

- Compression: Amount of time spent by each process performing compression on the data to be written.
- Meta Communication: Time needed to communicate information related to sizes and offsets.
- Data communication: Time spent on sending the data buffers from processes to the aggregators.
- Sorting: Time needed by the aggregators to calculate the compressed/uncompressed offsets and lengths before performing the write operation.
- Write: Time taken to write the data to the file system (aggregators only).

The results are shown in fig. 3. The results indicate that the time taken to write the data to the file system by the aggregators is the most dominant component in the overall execution of the collective I/O operation in these tests. Furthermore, it seems that the write operations are able to saturate our file system in these tests, since the write-time is roughly constant and independent of the number of processes used. The time spent on compression decreases with increasing number of processes, as each process gets a smaller portion of the matrix to compress.

From the time spent in communication operations, the meta communication time increases with increasing number of processes, while the data communication time decreases with the number of processes when using a fixed problem size. Both observations are in-line with the expected behaviour, since the meta communication is mostly based on collective operations (e.g. Allgather) which are typically becoming costlier with increasing process counts, while the data communication is mostly based on point-to-point operations, which become cheaper for a fixed problem size when spreading it out to more processes [20]. This effect is most pronounced with the largest process count of 529 where the meta communication time overtakes the compression and data communication time combined.

The sorting operation takes according to this analysis the least amount of time, is however increasing with increasing process counts. This is the expected behavior for a fixed problem size, since the same amount of data is split into more and smaller chunks that are then being used in the sort step.

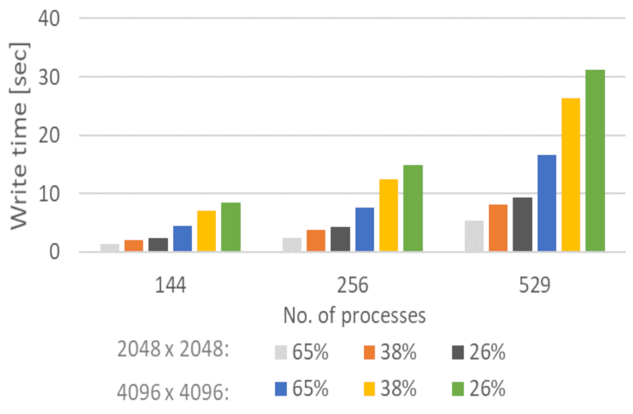

Fig. 4. Impact of the space saving factor on the execution time of a collective write operation.

2) Impact of Size Reduction: In the next set of tests we explore the impact of the compression ratio on the performance of the collective write operation. As discussed previously, the compression ratio can be adjusted by changing the size of the bucket used for populating the matrices, since the size of the bucket controls how many distinct floating point values are being used to populate the matrix: fewer values in the bucket lead to a higher compression ratio. For the subsequent tests, we used bucket sizes of 150 values, $1,000$ values and $5,000$ values. The resulting average space saving (or size reduction), which is defined as

$$1 - \frac{compressed\ size}{uncompressed\ size}$$

are approx. 65%, 38%, 26% respectively. The results shown in fig. 4 indicate as expected a positive correlation between space saving and performance, where the write time decreases as the space saving ratio is increased.

B. Real World Data Sets

In this set of tests, we evaluate the performance benefits of writing compressed output files using parallel I/O on real world data sets. The benchmark in this case uses a fixed size, uncompressed input file. It reads the input file using `MPI_File_read_all`, and writes the data using `MPI_File_write_all` requesting compression through the info object mentioned earlier. The test thus mimics a standard compression tool, except for the fact that all I/O operations occurring are using collective I/O. The times reported by the benchmark include both the reading of the uncompressed files and the writing of the compressed files, in order to compare the performance to the sequential snzip compression tool. Three data sets from various application domains are used to evaluate the benefits using data compression with MPI I/O.

a) Human Readable Text Data Sets: The first data set is a collection of English language ebooks from Project Gutenberg [3]. The generated data file consists of text files concatenated into a single file, which results in an approx. 13.5GB file. The two data sets used here, a 27GB and a 54GB file, are created by concatenating the 13.5GB file multiple times.

b) Genomic Data Sets: The second data set contains two FASTA formatted nucleotide sequences downloaded from the National Center for Biotechnology Information (NCBI). The two sequences used in our tests are the Larix sibirica (ID: 66971) [4], referred to as Genome 12 in our tests, since it is around 12GB in size, and Ambystoma mexicanum [1] (ID: 381), referred to as Genome 30 and 30GB in size.

c) Dark Sky Simulations Data Sets: The third data set is based on the Dark Sky simulations [2], which are a series of cosmological simulations meant to help study the evolution of a universe. The output files from these simulations contain multivariate information about particles and cosmological structures. The two data sets downloaded for our test are approx. 16GB and 77GB in size.

TABLE I
EXECUTION TIMES AND OF SEQUENTIAL AND PARALLEL COMPRESSION OF VARIOUS DATA SETS

Dataset	Size Reduction	Sequential	36 Procs	64 Procs
Gutenberg 27	40%	1142s	43.3s	30.2s
Gutenberg 54	40%	2282s	96.3s	61.7s
Genome 12	67%	338s	15.3s	11.9s
Genome 30	57%	1055s	42.0s	30.1s
DarkSky 16	24%	777s	24.5s	19.3s
DarkSky 77	32%	3175s	118.5	92.6s

Table I presents in the second column the average size reduction achieved with Snappy compression on each data set. Columns 3, 4, and 5, present the time it took to execute the sequential snzip tool on each input file, as well as the MPI code using the solution presented in this paper for performing the compression using 36 and 64 processes respectively. The results indicate significant performance improvements for the version using MPI I/O for compressing the data sets, with a speedup between 20 and 40 for the various benchmarks. The speedup values for each individual test case are also shown in fig. 5.

V. RELATED WORK

Data compression is a very actively researched field due to its applicability to modern demands of data processing and storage. For frequently used data, trading compression ratio for compression/decompression speed is a useful approach. LZ4 [13], for example, is a compression algorithm designed specifically for speed. Though the compression ratio for LZ4 is generally worse than other algorithms such as LZ77 [30], the decrease in processing time is usually much greater in comparison. Google's Snappy [8] compression library is related to the Lempel-Ziv-Oberhumer family of compression algorithms but is designed to incur low CPU usage and low execution time when compared to similar compression libraries such as LZO [28, 31]. There are also compression algorithms designed specifically for floating point data [29], which are notoriously difficult to compress with standard compression algorithms.

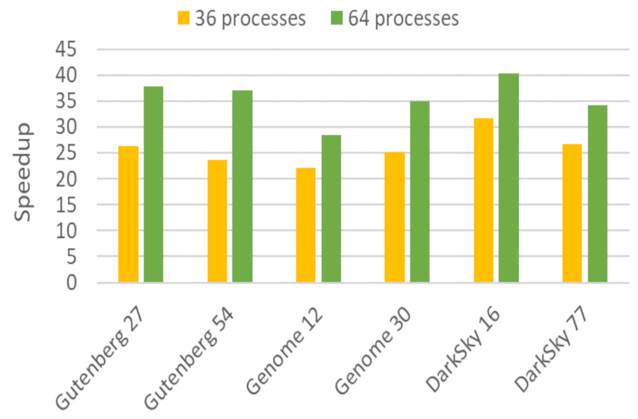

Fig. 5. Speedup of the parallel compression using MPI I/O over the sequential snzip tool.

More recently, researchers have also experimented with lossy compression techniques in scientific computing [32], since lossy algorithms often provide performance advantages compared to lossless algorithms.

In High Performance Computing, compression was used in numerous projects to reduce the communication volumes and times [15, 21, 23]. Since compressing and decompressing data buffers consumes (significant) resources, these techniques have however limited usability with decreasing network latencies and increasing network bandwidth on today's high-end computer systems.

Compression is however widely utilized in file I/O. In [15] the authors introduce data compression in collective I/O operation. Compression is however restricted to the communication phases of the collective I/O operation, and does not impact the file written to disk. The authors in [33] introduce the ability to compress data in ADIOS file I/O operations, and dynamically separate non-compressible and compressible data into separate streams. In contrast to the work presented in this paper however, ADIOS defines a proprietary file format and does not aim to write generic data files in a compressed format.

In [38] authors have introduced compression to file I/O operations for large scale simulations. The focus of the paper was however not on integrating compression into MPI I/O operations and understanding the impact on the interfaces and applications, but evaluating where the compression should occur in their architecture (helper cores, staging nodes, etc.). Similarly, the authors in [10] incorporate data compression into the I/O software stack, yet on the level above MPI I/O, i.e. invoking MPI I/O operations on compressed data.

Lastly, handling compressed data files is common in Hadoop [36] and Spark [37] applications. While these frameworks arguable perform parallel I/O, neither of the two software stacks manages shared-file parallel I/O, and thus sidestep many of the problems and challenges discussed in this paper.

VI. Conclusions

In this paper we discussed the key challenges, requirements, and solutions for supporting compressed data files in MPI I/O. The paper detailed an approach for handling individual read and write operations of compressed data files, and presented an extension to the two-phase collective I/O algorithm to support data compression. Furthermore, we implemented a parallel I/O library operating on compressed data files based on the Snappy compression library and the OMPIO parallel I/O framework. The paper evaluated the new functionality using multiple benchmarks using generated as well as real world data sets. The results obtained show significant performance improvements of up to 43% when writing a compressed output file using collective I/O vs. an uncompressed file. The code used for this paper is available on the webpages of the author research group (https:://pstl.cs.uh.edu/projects/ompio.shtml).

This work can be further enhanced by including support for other compression libraries such as bzip, and performing further tests on other data sets, as well as other storage and file systems. MPI furthermore supports the notion of of an *internal data representation* which can be used for performance improvements without having the interoperability requirement discussed previously, i.e. it is acceptable if the file cannot be read with another tool/software than the MPI library used to write the file. Omitting the *no gap in the output file* requirement and combining multiple compression algorithms by using e.g. compression algorithms that have been specifically designed for floating point values (e.g. [29]) could be used to define a highly performant internal data representation.

Acknowledgments. Partial support for this work was provided by the National Science Foundation under Award No. SI2-SSI 1663887. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Ambystoma mexicanum (dataset). https://www.ncbi.nlm.nih.gov/genome/?term=asm291563v2, last visited 2020-03-02.
[2] Dark Sky Simulations. https://darksky.slac.stanford.edu/simulations/ds14_a/, last visited 2020-03-02.
[3] Free ebooks - Project Gutenberg. https://www.gutenberg.org/wiki/Main_Page, last visited 2020-03-02.
[4] Larix sibirica (dataset). https://www.ncbi.nlm.nih.gov/genome?term=GCA_004151065.1&cmd=DetailsSearch, last visited 2020-03-02.
[5] Snzip, a compression/decompression tool based on snappy. https://github.com/kubo/snzip, last visited 2020-03-02.
[6] Snappy framing format. https://github.com/google/snappy/blob/master/framing_format.txt, 2013.
[7] An introduction to BeeGFS. https://www.beegfs.io/docs/whitepaper/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2018.
[8] Snappy: A fast compressor/decompressor. https://google.github.io/snappy/, 2018.
[9] S. A. Brown, M. Folk, G. Goucher, and R. Rew. Software for Portable Scientific Data Management. *Computers in Physics*, 7(3):304–308, May/June 1993.

[10] Huy Bui, Hal Finkel, Venkatram Vishwanath, Salma Habib, Katrin Heitmann, Jason Leigh, Michael Papka, and Kevin Harms. Scalable parallel i/o on a blue gene/q supercomputer using compression, topology-aware data aggregation, and subfiling. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 107–111. IEEE, 2014.

[11] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster 2011 conference*, page t.b.d, Austin, Texas, USA, 2011.

[12] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. OMPIO: A Modular Software Architecture for MPI I/O. In *in Y. Cotronis, A. Danalis, D. S. Nikolopoulus, J. Dongarra (Eds.) 'Recent Advances in the Message Passing Interface', Lecture Notes in Computer Science, vol. 6960*, pages 81–90. Springer, 2011.

[13] Yann Collet. Lz4: Extremely fast compression algorithm. https://lz4.github.io/lz4/, 2013.

[14] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[15] Rosa Filgueira, David E. Singh, Jess Carretero, Alejandro Caldern, and Flix Garca. Adaptive-compi: Enhancing mpi-based applications performance and scalability by using adaptive compression. *The International Journal of High Performance Computing Applications*, 25(1):93–114, 2011.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[17] Jean-Loup Gailly and Mark Adler. The gzip home page. https://www.gzip.org/, 2019.

[18] Jeff Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, volume 16, pages 559–564, 2004.

[19] Hierarchical Data Format Group. *HDF5 Reference Manual*, September 2004. Release 1.6.3, National Center for Supercomputing Application (NCSA), University of Illinois at Urbana-Champaing.

[20] Shweta Jha and Edgar Gabriel. Performance Models for Communication in Collective I/O Operations. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 982–991. IEEE Press, 2017.

[21] Jian Ke, Martin Burtscher, and Evan Speight. Runtime compression of mpi messanes to improve the performance and scalability of parallel applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 59–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Ketan Kulkarni and Edgar Gabriel. Evaluating Algorithms for Shared File Pointer Operations in MPI I/O. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume LNCS 5544, pages 280–289, Baton Rouge, USA, 2009.

[23] V Santhosh Kumar, R Nanjundiah, Matthew J Thazhuthaveetil, and R Govindarajan. Impact of message compression on the scalability of an atmospheric modeling application on clusters. *Parallel Computing*, 34(1):1–16, 2008.

[24] Robert Latham, Robert Ross, and Rajeev Thakur. Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. *Int. J. High Perform. Comput. Appl.*, 21(2):132–143, 2007.

[25] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[26] Grant Mackey, Saba Sehrish, and Jun Wang. Improving metadata management for small files in hdfs. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–4. IEEE, 2009.

[27] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard Version 3.1*, June 2015. http://www.mpi-forum.org.

[28] Markus Franz Xaver Johannes Oberhumer. LZO - a real-time data compression library. http://www.oberhumer.com/opensource/lzo/, 2008.

[29] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[30] Kunihiko Sadakane and Hiroshi Imai. Improving the speed of lz77 compression by hashing and suffix sorting. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 83(12):2689–2698, 2000.

[31] David Salomon. Dictionary methods. In *Data Compression The Complete Reference Fourth Edition*, chapter 3, pages 171–221. Springer, 2007.

[32] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 914–922. IEEE, 2015.

[33] Eric R Schendel, Saurabh V Pendse, John Jenkins, David A Boyuka II, Zhenhuan Gong, Sriram Lakshminarasimhan, Qing Liu, Hemanth Kolla, Jackie Chen, Scott Klasky, et al. Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 61–72. ACM, 2012.

[34] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS 99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182. IEEE Computer Society, 1999.

[35] Romvault webpage. Understanding 7z compression file format. http://www.romvault.com/Understanding7z.pdf.

[36] Tom White. *Hadoop: The Definitive Guide*, pages 100–109. O'ReillyMedia, Inc., second edition, October 2010.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[38] Hongbo Zou, Yongen Yu, Wei Tang, and Hsuanwei Michelle Chen. Improving i/o performance with adaptive data compression for big data applications. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1228–1237. IEEE, 2014.